12. Edit Distance

String Alignment

Knuth, in (Knuth, 1993), shows how to transform words into graph using a seven step ladder of one letter substitutions.

\[
\begin{align*}
\text{words} & \rightarrow \text{wolds} \\
& \rightarrow \text{golds} \\
& \rightarrow \text{goads} \\
& \rightarrow \text{grads} \\
& \rightarrow \text{grade} \\
& \rightarrow \text{grape} \\
& \rightarrow \text{graph}.
\end{align*}
\]

Imagine applications where the distance between strings could be useful.

Spell checkers provide a list of nearby words when a string is not found in the dictionary. DNA, the molecule of life, can be abstracted as strings over the alphabet

\[\text{DNA} = \{\text{A, C, G, T}\}\]

Geneticists study the similarities and differences in DNA among members of a species and between species.

The similarity of two strings can be measured by an edit distance: The cost of a sequence of edit operations that change one string into another. Many different edit operations and cost measures have been proposed. Most measures define a metric space.

Definition 15: Metric on Strings

Let \(\Sigma^*\) be the set of all strings over an alphabet \(\Sigma\). A metric \(d\) is a function

\[d : \Sigma^* \times \Sigma^* \rightarrow [0, \infty)\]

with the properties:

1. **Non-negativity:** \(d(s, t) \geq 0\) for all strings \(s, t \in \Sigma^*\)
2. **Zero-distance:** \(d(s, t) = 0\) if and only if \(s = t\)
3. **Symmetry:** \(d(s, t) = d(t, s)\)
4. **Triangle Inequality:** \(d(s, u) \leq d(s, t) + d(t, u)\)

The common edit operations are substitute, insert, and delete.
Definition 16: String Operations

Let \( \alpha \) and \( \beta \) be strings over \( \Sigma \). Let \( x \) and \( y \) be a character in alphabet \( \Sigma \). Define three operations:

- **Insertion**: \( \alpha \beta \mapsto \alpha x \beta \)
- **Deletion**: \( \alpha x \beta \mapsto \alpha \beta \)
- **Substitution**: \( \alpha x \beta \mapsto \alpha y \beta \)

For instance, to align \( \alpha = \text{ALGORITHM} \) with \( \beta = \text{ALGEBRAIC} \) this sequence of editing operations might be used, where \( s \), \( i \) and \( d \) stand for substitution, insertion, and deletion.

<table>
<thead>
<tr>
<th>A L G O - R - I T H M</th>
<th>A L G E B R A I C - -</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>i</td>
</tr>
<tr>
<td>i</td>
<td>s</td>
</tr>
<tr>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>

If each operation has a cost of 1, then the distance between these two string is 6.

Levenshtein distance is a commonly used metric on strings. It is defined iteratively based on string length. Let \( \lambda \) denote the empty string. Then, the distance from any string \( \alpha \) to \( \lambda \) is the length of \( \alpha \).

\[
d(\alpha, \lambda) = d(\lambda, \alpha) = |\alpha|
\]

Otherwise, let \( d_{\alpha, \beta}(i, j) \) be the distance between \( \alpha[1..i] \), the first \( i \) characters of \( \alpha \), and \( \beta[1..j] \), the first \( j \) characters of \( \beta \).

- If \( \alpha[i] \) is deleted in aligning \( \alpha[1..i] \) with \( \beta[1..j] \), this edit distance is the distance between \( \alpha[1..(i - 1)] \) and \( \beta[1..j] \) plus 1.

- If \( \beta[j] \) is deleted in aligning \( \alpha[1..i] \) with \( \beta[1..j] \), this edit distance is the distance between \( \alpha[1..i] \) and \( \beta[1..(j - 1)] \) plus 1.

- If \( \alpha[i] \) is substituted \( \beta[j] \) in aligning \( \alpha[1..i] \) with \( \beta[1..j] \), this edit distance is the distance between \( \alpha[1..(i - 1)] \) and \( \beta[1..(j - 1)] \) plus 1 if \( \alpha[i] \neq \beta[j] \). And, otherwise, it is just \( d((i - 1), (j - 1)) \).

The value of \( d(i, j) \) is the minimum over all of these values.

Definition 17: Levenshtein distance

Let \( \alpha \) and \( \beta \) be strings over \( \Sigma \). For \( 0 \leq i \leq |\alpha| \) and \( 0 \leq j \leq |\beta| \),
Define the edit distance between $\alpha[1..i]$ and $\beta[1..j]$ to be

$$
d(i, j) = \begin{cases} 
\max(i, j) & \text{if } \min(i, j) = 0 \\
\min \left( 
\begin{array}{l}
\begin{aligned}
d(i - 1, j) + 1 \\
d(i, j - 1) + 1 \\
d(i - 1, j - 1) + [\alpha_i \neq \beta_j]
\end{aligned}
\end{array}
\right) & \text{otherwise}
\end{cases}
$$

where $[False] = 0$ and $[True] = 1$ is the characteristic function.

### Problem 9: String Edit Distance Problem

**Decision Problem:** Given strings $s$ and $t$, is $m$ the minimum number of edits to transform $s$ into $t$?

**Function Problem:** Given strings $s$ and $t$, find one or more edit sequences that minimize the distance between these strings.

One alignment of

$$s = \text{TAGCTATCA} \quad \text{and} \quad t = \text{AGGCTATTA}$$

might look like this:

```
T A G - C T A T C A
- A G G C T A T T A
```

The table below shows the initial configuration when computing the minimal edit distance between

$$s = \text{TAGCTATCA} \quad \text{and} \quad t = \text{AGGCTATTA}$$

The rows and columns are labeled by the characters in the strings.

The $\lambda$ column shows the costs for inserting of $\text{TAGCTATCA}$ into an empty string. These costs are $d(i, 0) = i$ for $i = 0, \ldots, 9$.

The $\lambda$ row shows the costs for inserting $\text{AGGCTATTA}$ into an empty string. These costs are $d(0, j) = j$ for $j = 0, \ldots, 9$. 
Values can be computed along diagonals. The first computed value comes from comparing T and A.

\[
d(1, 1) = \begin{cases} 
    d(0, 0) (= 0) & \text{if } T = A \\
    \min \begin{cases} 
        d(0, 1) + 1 (= 2) \\
        d(1, 0) + 1 (= 2) \\
        d(0, 0) + 1 (= 1)
    \end{cases} & \text{otherwise}
\end{cases}
\]

Downward moves ↓ and horizontal moves → always cost 1. Diagonal moves ↘ cost 0 or 1 depending on whether the next characters match or not. The minimum of these costs is the new value.
Now, compute values in the next diagonal:

<table>
<thead>
<tr>
<th></th>
<th>λ</th>
<th>A</th>
<th>G</th>
<th>G</th>
<th>C</th>
<th>T</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>λ</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Next, compute $d(3, 1)$, $d(2, 2)$, and $d(1, 3)$.
The complete edit distance table is:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>G</th>
<th>G</th>
<th>C</th>
<th>T</th>
<th>A</th>
<th>T</th>
<th>T</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>λ</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>T</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Tracing back in the array reveals optimal alignments.
The optimal alignment has cost 3

\[
\begin{align*}
&T A G - C T A T C A \\
- &A G G C T A T T A \\
\text{d} &i &s
\end{align*}
\]

Here is a C implementation of edit distance.

**Listing 58: Iterative String Edit Distance**

```c
169a ⟨Iterative String Edit Distance 169a⟩≡
int editDist(char *s, int ls, char *t, int lt)
{
    int distances[ls][lt];

    (If either string is empty return 169b)
    (Initialize the first row and first column 170a)
    (For every pair of characters 170b)
    {
        (If characters match, use the previous distance 170c)
        (Otherwise, use the minimum distance 170d)
    }
    return distance[ls-1][lt-1];
}
```

If either string s or t is empty, return the length of the other, which translated to inserting its characters. The C-idiom is “if ls=0, then !ls is True.”

**169b ⟨If either string is empty return 169b⟩≡**
if (!ls) return lt;
if (!lt) return ls;

Initializing the first row and column has time complexity $\Theta(n + m)$.

\[\begin{align*}
&\text{Initialize the first row and first column} \quad 170a \\
&\text{for (int } i = 0, \text{ int } j = 0; \text{ i < m, j < n; i++, j++)} \\
&\quad \{ \\
&\quad \quad \text{distances}[i][0] = i; \\
&\quad \quad \text{distances}[0][j] = j; \\
&\quad \} \\
&\text{There are nm pairs of characters, assuming the source string } s \text{ has} \\
&\text{length } ls = n \text{ and target string } t \text{ has length } lt = m. \\
&\text{For every pair of characters} \quad 170b \\
&\text{for (int } i = 1; \text{ i < ls; i++)} \\
&\quad \text{for (int } j = 1; \text{ j < lt; j++)} \\
&\text{Testing for a match has complexity } O(1). \\
&\text{If characters match, use the previous distance} \quad 170c \\
&\text{if (s[i-1] == t[j-1])} \\
&\quad \text{distance}[i][j] = \text{distance}[i-1][j-1]; \\
&\text{And when a mismatch occurs, only a few table look-ups, compar-} \\
&\text{isons, and assignments are necessary.} \\
&\text{Otherwise, use the minimum distance} \quad 170d \\
&\text{else} \\
&\quad \{ \\
&\quad \quad \text{min = distance}[i-1][j-1]; \\
&\quad \quad \text{if (min > distance}[i][j-1])} \\
&\quad \quad \quad \{ \\
&\quad \quad \quad \quad \text{min = distance}[i][j-1]; \\
&\quad \quad \quad \} \\
&\quad \quad \text{if (min > distance}[i-1][j])} \\
&\quad \quad \quad \{ \\
&\quad \quad \quad \quad \text{min = distance}[i-1][j]; \\
&\quad \quad \quad \} \\
&\quad \quad \text{distance}[i][j] = 1 + \text{min}; \\
&\} \\
&\text{The performance of the edit distance algorithm is characterized by} \\
&\text{• Time complexity: } O(nm) \text{ to account for the nested for loops.} \\
&\text{• Space complexity: } O(nm) \text{ to account for storing the table.} \\
&\text{• Trace-back: } O(n + m) \text{ to construct the optimal alignment.} \end{align*}\]
Exercises

1. Fill in the optimal alignment table for seat and belt.
2. Fill in the optimal alignment table for park and spake.


Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms*. MIT Press, third edition. [page 9], [page 14], [page 18], [page 21], [page 31], [page 33], [page 39], [page 53], [page 93], [page 105], [page 135], [page 141], [page 163], [page 169], [page 178], [page 187], [page 193], [page 201]


Index

Haskell programming language, 20, 31, 44, 116, 137
Java programming language, 31

Ackermann function, 174
Axioms
Peano, 192
Bézout, Étienne, 47

C programming language, 30, 31, 42, 43, 55, 109, 152, 153
Calendar, 7
Cartesian product, 142
Catalan numbers, 156, 161
Computability, 191
Computable function, 28
Computer Language Benchmark Game, 30
Contradiction, 193
Dijkstra’s algorithm, 183
DNA, 146

Empty
String \( \lambda \), 61

Fibonacci, 44, 46, 142
Floating point
Machine epsilon, 50
Function
Ackermann, 174
Functional programming, 31
Gödel, Kurt, 192
Golden ratio, 142
Greatest common divisor, 47
Gutenberg Project, 84

Hash Tables, 93
Hofstadter, Douglas, 25
Kleene, Stephen, 53
Knuth, Donald, 30
Lamé,Gabriel, 44
\( \lambda \) calculus, 26, 28
Linear congruence, 47
Listing
C
Matrix Chain Multiplication, 158
Listings
C
A Minimum Algorithm, 135
Aho-Corasick, 89
Boyer-Moore Pattern Matching, 72
Brute-force Left-to-Right Pattern Matching, 56
Bubble Sort, 109
Build a Heap, 126
Counting Sort, 128
Direct Address Table Operations, 94
Edit Distance, 152
Find Set with Path Compression, 177
Greatest Common Divisor, 42
Hash Table with Chaining, 95
Heap Sort, 127
Heapify a List, 126
Inner Product, 155
Insertion Sort, 112
Knuth–Morris–Pratt Pattern Matching, 67
Maximum Subsequence Sum (Cubic), 34
Maximum Subsequence Sum (Linear), 35
Merge Sort, 118
Morris–Pratt Pattern Matching, 58
Partition about a pivot, 120
Quick Sort, 120
Radix Exchange Sort, 132
Radix Sort, 129
Randomized Partition, 136
Randomized Selection, 138
Rational Knapsack, 167
Selection Sort, 114
Shell Sort, 116
Union by rank, 177

Haskell
Ackermann function, 174
Apply a Function Repeatedly, 50
Brute-force Left-to-Right Pattern Matching, 57
Bubble Function, 108
Bubble Sort, 109
Extended Euclidean Algorithm, 48
Fold from the left, 37
Greatest Common Divisor, 44
Inner Product, 155
Insertion Sort, 112
Matrix Multiplication, 155
Maximum Subsequence Sum, 37
Merge Sort, 117
Merging Sorted Lists, 117
Newton’s Square Root Method, 51
Newton’s Square Root Recurrence, 50
Quicksort, 120
Randomized Partition, 137
Relative Error Convergence Test, 50
Sample Code Header, 31
Second of a pair, 36
Selection Sort, 114
Sorted Decision Problem, 106
Splitting a List, 117
Swapping head with Another Element, 137
Pseudocode
Activity Selection, 170
Bucket Sort, 130
Dijkstra’s Algorithm, 186
Kruskal Minimum Spanning Tree, 172
Prim Minimal Spanning Tree, 181
Lucas, Édouard, 44
Memorization, 141
Metric space, 152
Newton’s method, 13
Newton, Issac, 49
Noweb, 30, 55
Object-oriented programming, 31
Pattern Matching, 53, 54
Boyer-Moore, 71
Brute Force, 55
Knuth–Morris–Pratt, 67
Morris–Pratt, 58
Right-to-Left Brute Force, 70
ψ, 142
Presburger arithmetic, 191
RAM, 26, 29
Sequence, 19
Lucas, 44
Sets
Cartesian product, 142
Simpson’s rule, 51
Sorting Algorithms
Bubble Sort, 107
Bucket Sort, 129
Counting Sort, 128
Heap Sort, 124
Insertion Sort, 111
Merge Sort, 116
Quick Sort, 119
Radix Sort, 129
Selection Sort, 114
Shell Sort, 115
Time complexity, 127
String
Border, 61
Period, 61
Prefix, 61
Suffix, 61
Syllabus, 5
Theorems
Bezout’s, 47
Euclidean division, 41
Lamé’s, 45
Taylor’s, 49, 51
Turing
Machine, 9, 26, 28, 192
Wikipedia, 201
xkcd
Algorithms by Complexity, 25
Calendar of meaningful dates, 7
Haskell, 30
Ineffective Sorts, 105
Traveling Salesman Problem, 141